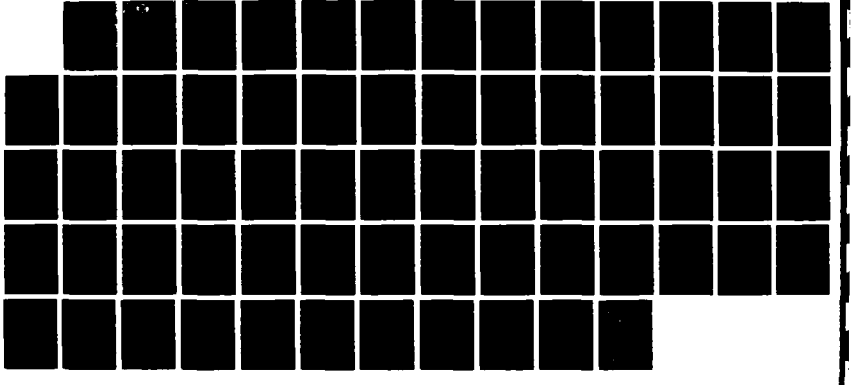
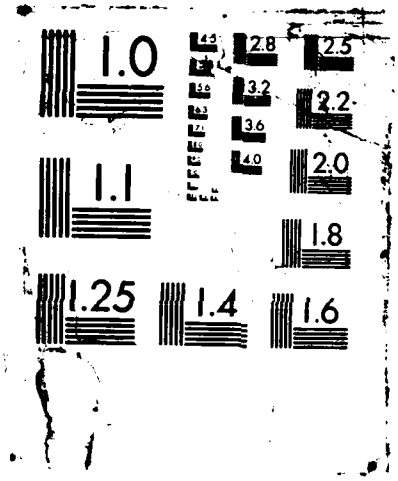
AD-A191 446 TUNING A MAJOR PART OF A CLUSTERING ALGORITHM(U) 1/1
PRINCETON UNIV NJ DEPT OF STATISTICS K M HANSEN ET AL.
FEB 88 TR-294 ARO-23360.8-MA DAAL03-86-K-0073
UNCLASSIFIED F/G 12/3 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
AD-A191 446	5. MONITORING ORGANIZATION REPORT NUMBER(S): ARO 23360.8-MA
6a. NAME OF PERFORMING ORGANIZATION: Princeton University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code): Princeton, NJ 08544	7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U. S. Army Research Office; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAL03-86-K-0073
8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Tuning a Major Part of a Clustering Algorithm

12. PERSONAL AUTHOR(S): Katherine M. Hansen and John W. Tukey

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	February 1988	32

16. SUPPLEMENTARY NOTATION: The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Clustering Algorithm, Algorithms, Clustering Procedures, Gaussian Samples

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Most proposals for clustering algorithms have been based on introspection. Few proposed algorithms have had their performance studied. ~~Our~~ This approach involves (a) striving to avoid comparing distances on remote parts of the data (because metrics deserve only minimum trust), and (b) using a stochastically-defined test bed to measure, and where possible understand, the performance of an evolving algorithm, with the intent of using our understanding to modify it in such a way as to improve its performance.

Abstract continued on reverse side

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

This test bed involves 3 circular Gaussian samples, of size 50 each, centered at the vertices of an equilateral triangle of side t. In its use we assume that a 3-group answer is being sought. Thus only concerned with a part of the clustering process.

Our early algorithms begin to misbehave in the range $5 \leq t \leq 7$. Our successive steps of improvement work at smaller and smaller t. The last version we have tried still performs usefully (median misclassification about 16%) at $t = 2.7$, where knowledge of three populations would only let us hold misclassification to a median of 13.3%.

Comparison (by Kaye Basford) with a Gaussian maximum likelihood algorithm on the same set of triple samples shows only slightly better performance than for our algorithm.

Tuning a major part of a clustering algorithm

by

Katherine M. Hansen and John W. Tukey

Princeton University
Fine Hall
Washington Road
Princeton, NJ 08544

Technical Report No. 294
Department of Statistics
Princeton University 08544

February 1988

Prepared in connection with research at Princeton University
sponsored by the Army Research Office (Durham), DAAL03-86-K-0073.

Tuning a major part of a clustering algorithm

Katherine M. Hansen and John W. Tukey

Princeton University
Fine Hall
Washington Road
Princeton, NJ 08544

ABSTRACT

Most proposals for clustering algorithms have been based on introspection. Few proposed algorithms have had their performance studied. Our approach involves (a) striving to avoid comparing distances on remote parts of the data (because metrics deserve only minimum trust), and (b) using a stochastically-defined test bed to measure, and where possible understand, the performance of an evolving algorithm, with the intent of using our understanding to modify it in such a way as to improve its performance.

Our test bed involves 3 circular Gaussian samples, of size 50 each, centered at the vertices of an equilateral triangle of side $t\sigma$. In its use we assume that a 3-group answer is being sought. Thus we are only concerned with a part of the clustering process.

Our early algorithms begin to misbehave in the range $5 \leq t \leq 7$. Our successive steps of improvement work at smaller and smaller t. The last version we have tried still performs usefully (median misclassification about 16%) at $t = 2.7$, where knowledge of three populations would only let us hold misclassification to a median of 13.3%.

Comparison (by Kaye Basford) with a Gaussian maximum likelihood algorithm on the same set of triple samples shows only slightly better performance than for our algorithm.

The steps of improvement will be discussed.

Prepared in part in connection with research at Princeton University sponsored by the Army Research Office (Durham), DAAL03-86-K-0073.

Tuning a major part of a clustering algorithm

Katherine M. Hansen and John W. Tukey

Technical Report No. 294
Princeton University
Fine Hall
Washington Road
Princeton, NJ 08544

1. Introduction.

Since clustering ought to be an aspect of data analysis, we ought to plan to take an empirical evolutionary approach to its conduct. This is done here for a major part of the clustering problem: how to cluster *given* the number of clusters to be sought.

PART A: Background and final performance

2. Clustering.

Some have argued that the purpose of clustering a set of points is to *prove* that points belong to different groups. We have taken the opposite extreme, expecting to have misclassification - - real problems are hard - - but striving to minimize it.

Most clustering procedures can be persuaded to give rise to a variety of answers; one putting all the points in one group, through putting them in two groups, in three groups, and so on up to each point its own group. When, as is most often the case, these subgroupings are nested - - e.g. every group of the set of 4 groups is contained in *one group* of the set of three - - or of two - - the

Prepared in part in connection with research at Princeton University sponsored by the Army Research Office (Durham), DAAL03-86-K-0073.

clustering is usually called *hierarchical.* Knowing where to stop the gathering-together process is obviously very important. However, we have not tried to work on this problem. Our work has focussed on doing a good job of making a prechosen number of clusters, specifically 3.

Many procedures operate by adding links - - connections of one vertex (one data point) to another - - step by step, often one at a time. Once two data points have been joined by a chain of links, there is no need to join them further. Thus most methods grow what are combinatorially *trees,* but what *look* - - in 2 and 3 dimensions where we can look - - like *vines.* We shall call sets of vertices linked in a way that includes no cycles *vines.*

Most clustering procedures begin with either an overall criterion or a next-link-to-be-added rule. These criteria or rules typically come down from on high, like the famous tablets of stone. One can then only illustrate their performance. Our work has focused on *evolving* an algorithmic procedure, step by step, by looking for weaknesses of performance in the current algorithm, and then trying to find helpful modifications.

Most clustering procedures use single criteria or single rules. We have found it important to use successive steps that use quite different rules or criteria in different steps.

All clustering procedures use distance measures, or some sorts of surrogate. We have confined ourselves to starting with Euclidean ($L^{(2)}$) distance, but have emphasized the rank of the nearness of other points to a selected point. Thus each link, say AB, has two such ranks, one for how near

(rankwise) a neighbor B is to A and one for how near (rankwise) a neighbor A is to B.

3. Test bed.

Our basic tool in evolving our procedures has been triple samples, of 50 each, from 3 circular Gaussian distributions, centered at the vertices of an equilateral triangle of side $t\sigma$. We have found 10 triple samples, for each t, enough to guide us in making the next step of improvement. More would, of course, be needed if one wanted more precise assessment of algorithm performance. We have drawn fresh sets of 10 triple samples for each t, so that performance consistency for nearby t offers independently supportive evidence. We have so far used, successively, t=7, 5, 4.5, 4, 3.7, 3.2, 2.9 and 2.7.

We judge the performance of any procedure on a triple sample by the number of points misclassified - - specifically by the number of minorities, see below. The performance on a set of 10 triple samples is then measured by the median, or perhaps the mean, of the number of misclassifications.

4. Performance of the currently preferred procedure.

*** test bed ***

We now describe briefly the performance of the procedure summarized in Section 16, which is our currently preferred choice, both on the test bed and on some real data.

Two extremes of the 10 realizations of the test bed for t=2.7, one easy and one hard, are shown in exhibit 1. The numbers of misclassifications are 16

and 73, respectively. If we knew the 3 population distributions exactly, there would be an optimum partition of the plane into 3 120^o sectors. Classification by these sectors would have made 16 and 26 misclassifications, respectively. (The median (mean) numbers of misclassifications, over the 10 realizations are 25.5(29.7) for the procedure, and 20(21.3) for infinite knowledge.)

exhibit 1 about here

With only 3 samples of 50 each, we come, in median, within 5 of the number of misclassifications corresponding to knowing the populations (but not how many observations come from each). Any substantial further improvement seems unlikely.

In cooperation with Dr. Kaye Basford, the same sets of triple samples have been analyzed using a Gaussian maximum likelihood procedure (Basford and McLachlan, 1985), which does slightly better than ours (and, of course, still not as well as population knowledge). Since real data is not likely to be exactly Gaussian, and since no Gaussian assumption was explicit in our development (or visible in the final procedure) it is reasonable to hope that our procedure will at least equal the performance of Basford and McLachlan's on real data.

* real data *

Our main test bed is conveniently 2-dimensional (3 population centers define a plane), but our algorithms make no explicit use of this fact. Thus we have no difficulty applying them to more dimensional data.

Our first real example is the well-known *Iris* data, due to Edgar Anderson

(1935) and used by R. A. Fisher (1936) in his basic paper on the discriminant function. In the present context, we neglect the information about species (there were 50 observations on each of 3) and use only the 4 coordinates (petal and sepal length and width) of each blossom. We find only 23 misclassifications (all of, course, involving the two most similar species). Standard methods give 31, 50 and 47, respectively.

Our second real example is geochemical, involving analyses of chemical results for samples from 3 carbonate microfacies (here labelled 1, 2, and 6). Several views of the data are given in exhibit 2. The first of 12 variables represents the weight percent of insoluble residue for each sample. The remaining variables represent concentrations of eleven elements. For this data, our procedure, given an exogenous choice of three as the number of clusters, makes only 3 misclassifications. (Earlier attempts with classical clustering procedures were quite unsatisfactory; 17 misclassifications for complete linkage and single linkage, 18 for average linkage. Ward's method gave 7 misclassifications.) (Data for the geochemical example provided by the courtesy of Ms. Ruth Strauss.)

exhibit 2 about here

5. What we have not yet done.

By limiting our more careful systematic studies to test beds involving

- exactly 3 populations
- each circular in shape

_ symmetrically placed

_ in 2 dimensions

_ Gaussian in shape

we have clearly only begun a careful investigation.

We have done a little to explore (a) a similar test bed in 5 dimensions, (b) a similar 2-dimensional test bed with 3 populations of different variances and (c) a 2-dimensional test bed with 3 populations of different shapes. Our results have been sketchy but encouraging in all cases, (see Sections 18 to 20) as have very preliminary looks (not discussed further here) at ways to "squeeze down" unneeded dimensions.

What can be done, once an overall clustering has been completed, to look at pairs - - or other small groups - - of clusters separately, with a view to polishing the clustering, is uncertain.

Clearly much further exploration would be helpful.

PART B: Evolution of our procedures

6. Initial choices.

The shakiest part of any clustering procedure is the choice of the metric. No one has seen any way to avoid its use, directly or indirectly. Equally, no one has seen any way to use the final clustering as a basis for improving the choice. Alas, alack-a-day!

The sort of difficulty that can arise is well shown in exhibit 3, where a

classical clustering procedure is run on data naturally falling into 3 groups of differing spread.

exhibit 3 about here

On careful examination, however, while it is apparent that while we *must* use the metric locally, it is far from obvious that we need use it globally. By focusing on the nearness-rank of each link from both ends, we focus on only relatively local uses of the metric.

While we do use the original metric for tie-breaking, it seems unlikely that such uses have substantial effects on the outcome of our procedure.

What we would like to have - - and have only begun to approach - - would be a near invariance of result when, for instance, one coordinate, y, $1 \leq y \leq 10$, were replaced by e^y. It seems likely that further steps on such a direction must come from procedures to reconfigure coordinates rather than from clustering procedures themselves.

7. Nearest neighbors and total minorities.

Our methods are all based on *ranks of neighborliness,* equal to j if a point y is point x's j^{th} nearest neighbor, that is, when exactly $j-1$ points are closer to x than is y. Note that this is not necessarily a symmetric relationship, i.e. x need not be y's j^{th} nearest neighbor. We measure nearness using the Euclidean $L^{(2)}$ norm.

The goodness of our cluster solutions on the triangular Gaussian test beds is measured by a *total minorities* criterion, which is computed as follows:

- run the algorithm until exactly three clusters remain
- count the number of points in each output cluster coming from a minority subswarm (from either subswarm other than the single subswarm occurring most frequently)
- add up these counts, to find the total minority count.

For example, with the following output:

	subswarm A	subswarm B	subswarm C	Minorities
cluster # 1	45	7	2	9
cluster # 2	3	35	0	3
cluster # 3	2	8	48	10
				22

the total minority count is 22.

8. Minrank and maxrank - the first approach

Our initial algorithms are based on some simple properties of links between points. Consider a link joining two points, x and y. The link may be classified according to two ranks. If point y is point x's R_x^{th} nearest neighbor and point x is point R_y^{th} nearest neighbor, then define the following:

$$\text{minrank} = \min\{R_x, R_y\}$$
$$\text{maxrank} = \max\{R_x, R_y\}$$

It is these quantities which we plan to substitute for distance. With either minrank or maxrank as our basic unit of link length, we will have a substantial problem with ties. Both here and later we shall break ties with Euclidean distance; however we will soon move to a criterion which has much less

difficulty with ties.

Thus, if we are using maxranks, the first link to be added will be the shortest of all links having a maxrank of 1.

Both the minrank and maxrank methods were tested on the test beds with $t=5$ and $t=7$. Results are shown in exhibit 4. The performance of both algorithms is fair over most trials at $t=7$, and is poor at $t=5$. It seems that minrank does a better job than maxrank.

exhibit 4 about here

However, when we plot the actual vines, we have reason to suspect that using maxranks may be the more promising approach. This is due to the simple character of many of the maxrank failures.

exhibit 5 about here

Exhibit 5 shows a poor maxrank solution for $t=7$. For this data set (total minorities = 49), we have been left with one very small cluster out of the three we have asked for. This is not surprising since a rather isolated point is likely to produce only links with large maxranks. Such a point will be added to a vine late in the selection process. This problem may be easier to diagnose and correct than the problems with the minrank algorithm.

9. The basic criterion.

We want to force links to isolated points to be added to vines earlier in the process. One way of identifying such links is by noting the sizes (counts) of the two vinelets (protoclusters) that would be joined by adding the link. For an

isolated point the lesser of these counts will be one, so that we might consider a link-criterion of the following form:

basic criterion = (5 + maxrank)(1 + ln (lesser of counts)).

At each step we add the link with the lowest criterion value, again breaking ties with the Euclidean distance between the two points joined by a link. Note that since cluster counts increase at each step, the basic criterion value for each edge must be recomputed regularly.

exhibit 6 about here

Exhibit 6 shows the performance of this technique for $t=4$, 4.5, and 5. With $t=5$, performance is quite good; results for $t=4$ and 4.5 are not as satisfactory. Note, however, that at $t=4$ the performance of the basic criterion surpasses that of the minrank and maxrank algorithms at $t=5$.

At this part, we tried using minrank and (arithmetic and geometric) meanrank in place of maxrank in the definition of the basic criterion. Different values of the constants were tested. These adjustments did not give improved results.

As before, we turn to pictures of vine-clusters to reveal configurations giving poor results. Exhibit 7 shows the worst of the ten trials for $t=4.5$; this gave 44 total minorities. Clearly the difficulty lies with the filament running horizontally across the picture. We need to find ways to improve on such vines.

exhibit 7 about here

10. Using core points - the basic method.

We would like to be able to emphasize certain "core points" which lie near the centers of each subswarm. If we can identify such points - - we shall suggest three ways of doing so - - we can build the framework of a cluster solution based on them.

Suppose that we have a list of about 50 core points, ignoring for the present how this list was obtained. Then we can use the following basic core-growing technique:

- Start from the beginning with this reduced set of points, using the basic criterion algorithm to add links until exactly three clusters, involving only the 50 points, are left.
- Treat this structure as the beginning of a vinery for all 150 points.
- Links are now added - - between single points and pre-formed groups of size $\geq$ 2 (in practice, usually $\geq$ 15) until only 3 clusters remain - - in the order determined by the basic criterion, which in this case is equivalent to maxrank.

The three core-point vines serve as foundations to which the other points are added.

11. Identifying core points I - high-order and vine neighbors.

One property of points in a vine is their *order*, or the number of links joining them to other points. The central part of the long filament running across the picture of Exhibit 7 is unbranched - - that is, all of the points are of

order $\leq$ 2. This suggests that we define core points based on properties related to order. Our first type of core points was identified as follows:

- run the basic criterion algorithm until a complete ($n-1$ links) vine has been generated,
- from this vine, choose as core points those points that

(a) are of order $\geq$ 3 or

(b) are a vine-neighbor of such a point AND are on an arc of links joining one point of order $\geq$ 3 to another such.

The points satisfying these criteria for the data of exhibit 7 are in exhibit 8. These points do seem more clearly separated into three parts.

exhibit 8 about here

Results for the basic core-cluster technique with core points satisfying the above criteria for the $t=4$ and 4.5 data sets are given in exhibit 9.

exhibit 9 about here

We see a small improvement over the basic criterion method at $t=4$; this was not repeated at $t=4.5$. Examining pictures of core clusters indicates that inclusion of vine-neighbors of high order points may be responsible for some of the problems. It is this observation that leads us to the next method of describing core points.

12. Identifying core points II - high-order.

We now identify only points of order $\geq$ 3 as core points. The basic core-

cluster technique applied to these points for $t=4$ and 4.5 gives the results summarized in exhibit 10. A considerable gain over both the basic procedure and the first core-point procedure has been achieved.

exhibit 10 about here

13. Identifying core points III - local density.

A third way to select core points is to choose points lying in regions of high point-density, since these regions are likely to represent cluster centers. However, we do not wish to use a global measure of density directly - we want to use *local* properties to identify core points. To do this takes a several-step procedure. The approach we use is as follows:

- Build a complete spanning vine (combinatorially a minimum spanning tree) using the basic criterion.
- Find the density (global) at each point, defined to be (distance to 10^{th} nearest neighbor)$^{-2}$ (other exponents might be appropriate for dimensions other than 2); other constants than 10 would be appropriate for other total numbers of data points.
- Compute for each point on the vine the following:

$$\text{logdens} = \log(\text{density}).$$

- Smooth logdens on the vine as follows:
  - For a point of order 1, do not change logdens.
  - For a point of order 2, replace logdens by the median of logdens values at the original point and at the two directly linked points.

- For a point of order ≥ 3, replace logdens by the second-highest of the logdens values at the original point and at each directly, linked point.

- Continue smoothing cycles until no further changes take place.

- Find all local maxima - - points which are not linked to higher density points - - either directly or through points of equal logdens.

- For every point, record the difference in logdens (log ratio of densities) from the highest local maximum that can be reached monotonically along the vine. (A local max can be reached monotonically along the vine from a point if an arc of links connects the two and if the logdens values at the points along the arc increase weakly monotonically towards the local maximum.) We use small logdens differences (zero for local maxima) as our measure of high local density.

- We want about 1/3 of the total points to be identified as core points. We will choose all of the local maxima and will select others in order of increasing logdens difference. Due to the smoothing of logdens values on the vine, it sometimes occurs that several points with equal logdens differences are tied for the 50th position on the list. In this case, none of the tying points are chosen as core points, and the total number of core points is less than 50.

---

exhibit 11 about here

Using the above approach to select 50 core points for each data set at $t=4$ and 4.5, we obtain the results given in exhibit 11. The algorithm represents an

improvement over the basic-criterion algorithm, and seems to be comparable in performance to the high-order algorithm. Exhibit 12 gives results for the local density and high-order algorithms with the $t=3.2$ and 3.7 data. For these data sets, the local density method generally gives lower minority counts.

exhibit 12 about here

If all the data points adhere closely to a smooth curve - - which might be a straight line - - then the complete vine will have no points of order ≥ 2 and there are no high order core point so the first two approaches fail. However, the local density approach will still apply!

14. Basic bisector polish.

Examining pictures of vine-clusters formed from core points, we discover a pattern common to many of the failures of the local density and high-order algorithms. Exhibit 13 is a good example.

exhibit 13 about here

The core points of each cluster are centered in an appropriate location, yet branches to outlying core points bring in data points from other subswarms. We would like to use information about the location of the core-point clusters without allowing distant points to join in the cluster. The easiest way to do this is to give up our reliance on handling links one at a time, in fact to limit the use of links to the core-point clusters. We suggest the following approach:

- Grow vines of core-points using the basic criterion, continuing until exactly three core-clusters remain (can a good stopping rule be found

here?)

- Calculate the centers of gravity of the points constituting each core-*cluster.*

- Calculate the Euclidean distance from each data point to each center of gravity. A data point is assigned to the cluster corresponding to the nearest center of gravity. (Note that some core points might be re-assigned to clusters other than their initial core-cluster.)

For the case of three clusters, the algorithm described above is equivalent to separating the clusters by the perpendicular bisectors of the triangle with vertices at the three centers of gravity. Exhibit 14 shows the bisectors for the data of exhibit 13, using the local density core points. The number of minorities has been reduced from 32 to 12.

exhibit 14 about here

15. Bisector polish - notes, results, and comments.

We can use the basic bisector polish algorithm with either the high-order or the local density core points. We can also *iterate* the process by running the basic bisector polish algorithm and then computing the centers of gravity of each of the completed clusters. Points are then re-classified according to their distance from each of the iterated centers.

PART C: Performance of current algorithm

16. Comparative performance.

Results from the basic bisector polish and the iterated bisector polish algorithms for $t = 2.9$ are given in exhibit 15.

exhibit 15 about here

As we can see, bisector polishing represents an improvement over both the high-order and local density methods. Exhibit 16 gives the results of bisector polishing when *population means are used in place of core-point centers of gravity.* These are the best fixed partitions that can be chosen. Their performance gives a lower bound to the number of total minorities we might reasonably expect.

exhibit 16 about here

A brief comparison with standard methods of clustering is given in exhibit 17.

exhibit 17 about here

17. Current preference.

Our current recommendation then is the following overall algorithm:

A) Use the basic-criterion algorithm (Section 9) - - with its step-by-step insertion of links - - to form a single spanning vine containing all data points.

B) Use the local-density algorithm (Section 13) - - with its near-neighbor density, vine smoothing, and differencing - - to identify core points.

C) (Here would be a logical place for a step to identify how many clusters we want.)

D) Use the basic-criterion algorithm (Section 9) again, this time on the core points only, to form 3 clusters.

E) Starting with the centers of gravity of these 3 clusters, use the iterated bisector algorithm (Sections 15, 16) to assign every point to one of the three clusters.

Notice the presence of 4 or 5 steps, using at least three wholly different algorithms. Our preferred procedure is not simple, but it seems to be effective.

Notice also that, while steps (A), (B), and (D) - - presumably together with (C), when such becomes available - - do fairly well in focusing on local comparisons of distances, step (E) compares much larger distances. A plausible place for a further step (after E) would involve looking even more carefully at pairs of adjacent clusters with the intent of improving their separation.

PART D: Sketches of further explorations

18. 5-dimensional test bed.

We have looked briefly at 5-dimensional test beds constructed as follows:

- two of the dimensions represent the familiar circular Gaussians, with 3 subswarms centered on the vertices of an equilateral triangle of side $t\sigma$

- the remaining 3 dimensions are independently sampled from the standard Gaussian distribution.

We tested the basic and iterated bisector polish algorithms on 10 realizations of the 5-dimensional structure at each of t = 3.2, 2.9, and 2.7. In terms of the median number of misclassifications, the basic bisector polish algorithms were inferior to the corresponding iterated bisector polish algoirthms. Exhibit 18 shows the results for the iterated bisector polish algorithms.

19. Unequal-variance test bed.

All of the core-point algorithms were tested on a series of synthetic data sets containing 3 circular Gaussian subswarms of unequal variance. As before, the subswarms are centered at the vertices of an equilateral triangle. the triangle has sides of length $(\frac{1}{2}\sqrt{2} + \frac{1}{2}\sqrt{3})\lambda\sigma$, where the first subswarm has variance σ^2, the third has variance $3\sigma^2$. We generated 10 such data sets at each of λ = 3 and λ = 2.7.

The iterated bisector polish algorithms out-performed both the basic core-point and bisector polish methods. Results for the iterated bisector polish algorithms are summarized in Exhibit 19. Exhibit 20 shows the results from classification by maximum likelihood and the population bisectors. Each of the densities was completely specified for the maximum liklihood calculations; only the means were used to calculate population bisectors.

20. Unequal-shape test bed.

We generated series of test beds containing 3 2-dimensional Gaussian subswarms of unequal shape. The subswarms lie along an horizontal line segment of length $2t$, we centered at each endpoint of the segment; the third subswarm is centered at the midpoint. The central subswarm has standard deviation σ in the direction parallel to the segment and 3σ in the direction perpendicular to the segment. The subswarms at the endpoints of the segment are circular Gaussians with standard deviation σ.

We generated 10 test beds for each of $t=3.7$ and 3.2. Exhibit 21 shows two realizations of the unequally-shaped test beds. Results for the iterated bisector algorithms are given in Exhibit 22.

REFERENCES

Anderson, Edgar (1935). "The irises of the Gaspe Peninsula," *Bull. Amer. Iris Soc.* 59: 2-5.

Anderson, Edgar (1936). "The species problem in *Iris*," *Ann. Mo. Bot. Gdn.*

Basford, K. and McLachlan, G. J. (1985). "Likehood estimation with normal mixture models," *Appl. Statistics*, 34: 282-289.

Fisher, R. A. (1936). "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, 7: 179-188.

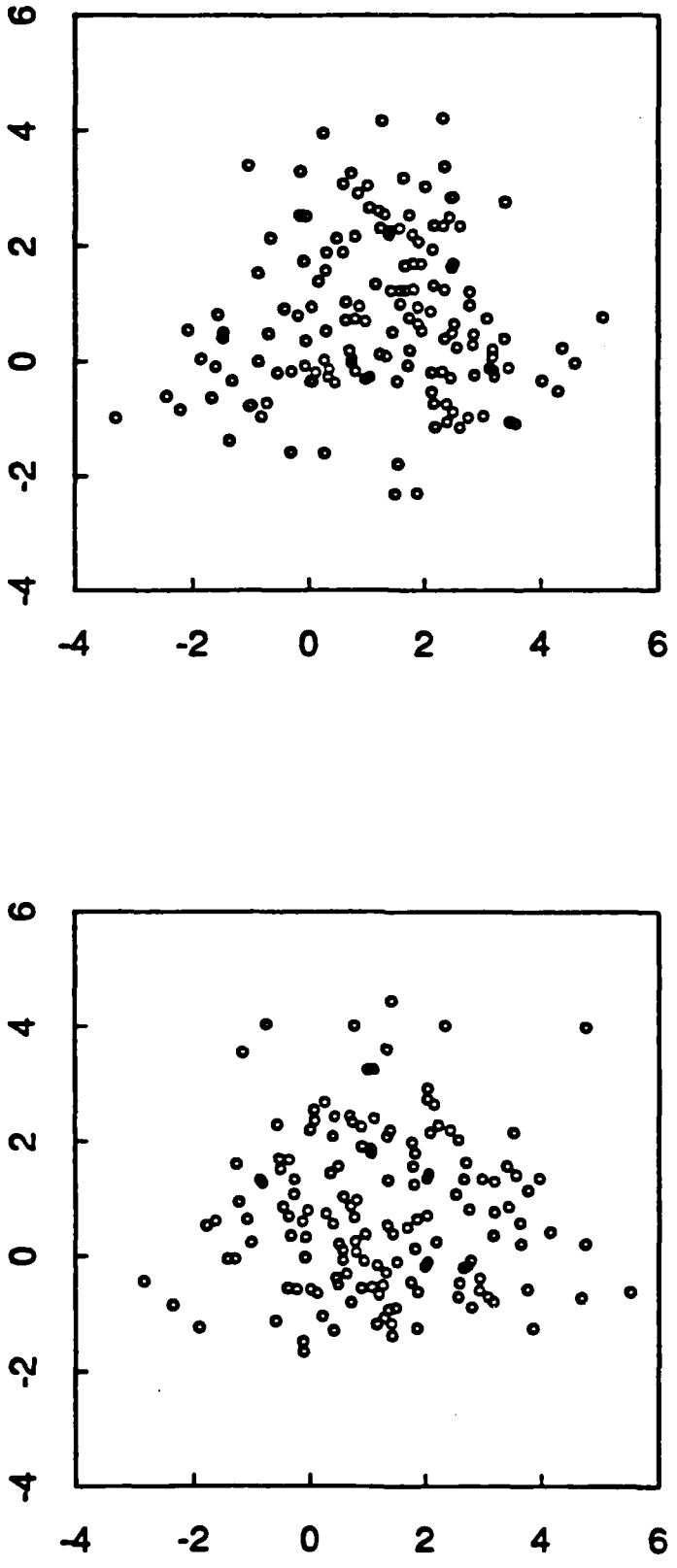


EXHIBIT 1: Two instances, one easy, one hard, of test bed triple samples (not separately identified) for t = 2.7. The preferred procedure gave 16 misclassifications for the upper set and 73 for the lower set.

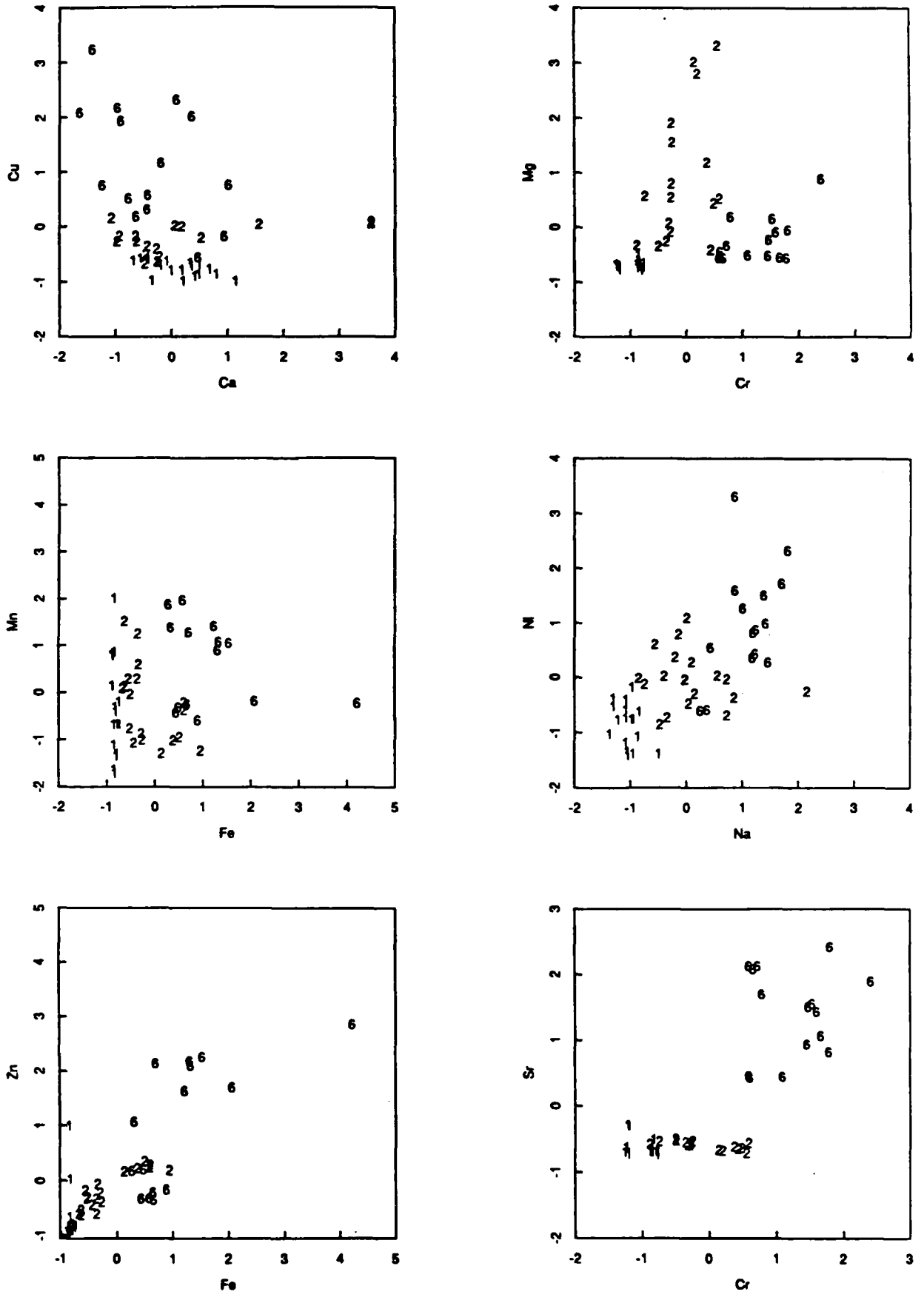


EXHIBIT 2: Six views of the geochemical data. (Plotting character = facies number.)

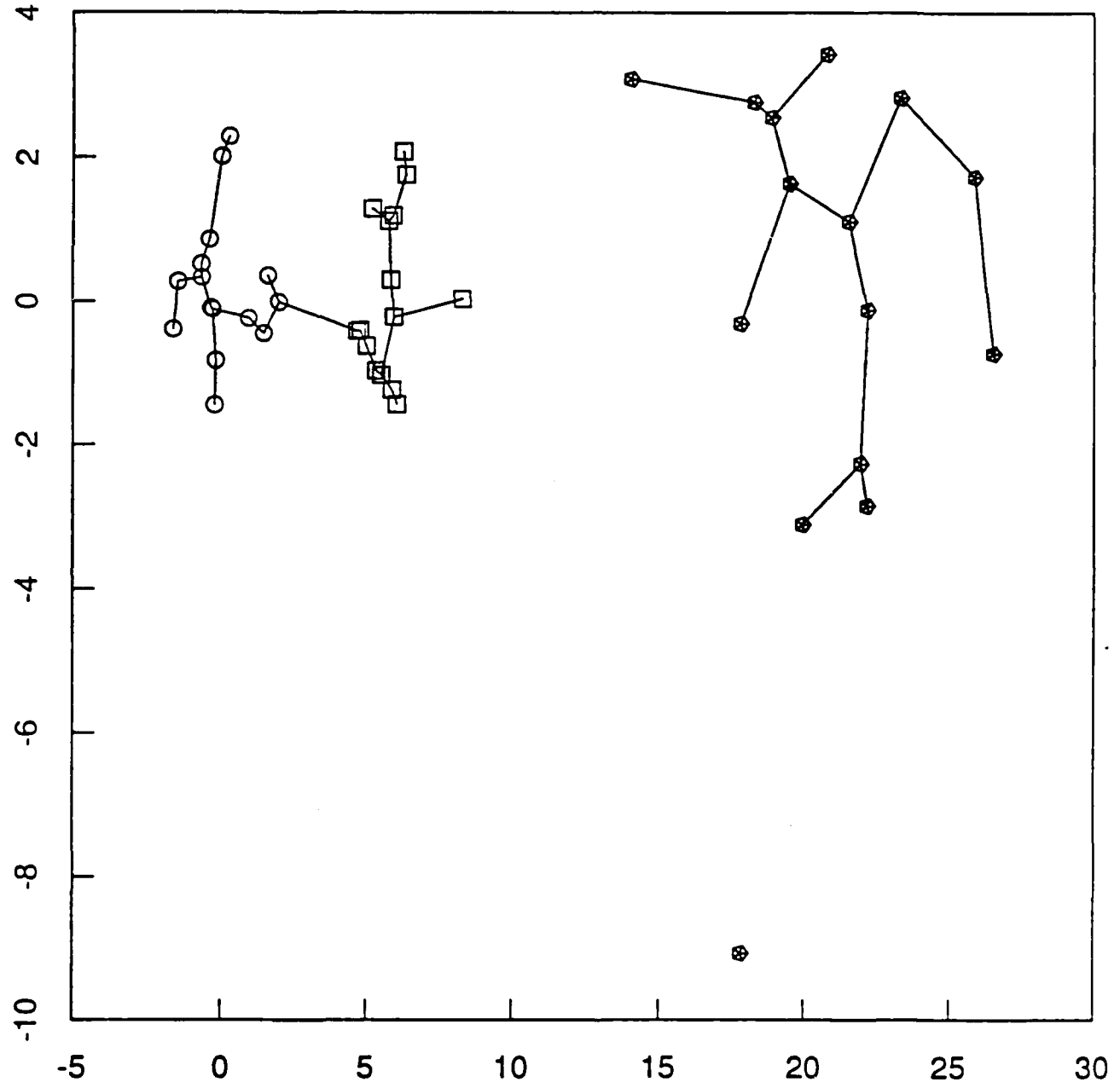


EXHIBIT 3: Result of running the single-linkage clustering algorithm on a data set containing three subswarms of unequal variance. The first two subswarms, plotted as circles and squares, have been merged into one cluster. The third subswarm, with greater variance, has been split into two clusters, one containing only the single point at lower right.

EXHIBIT 4: Some sample results on total minorities for the first approach.

TOTAL MINORITIES				
	maxrank		minrank	
	t=7*	t=5**	t=7***	t=5****
Median	25h	63	0	47
uHinge	49	97	50	50
Max	53	98	86	92
(Mean)	25.3	66.9	18.9	39.7

* Actual total minorities: 0,0,0,0,2,49,49,49,51,53

** Actual total minorities: 1,50,51,51,51,75,97,97,98,98

*** Actual total minorities: 0,0,0,0,0,0,2,50,51,86

**** Actual total minorities: 1,2,10,38,47,47,50,50,60,92

Note: We use "h" for "and a half" in reporting medians or hinges, where this is the only kind of non-integral value that can appear; we use ".5" for "and a half" in reporting means.

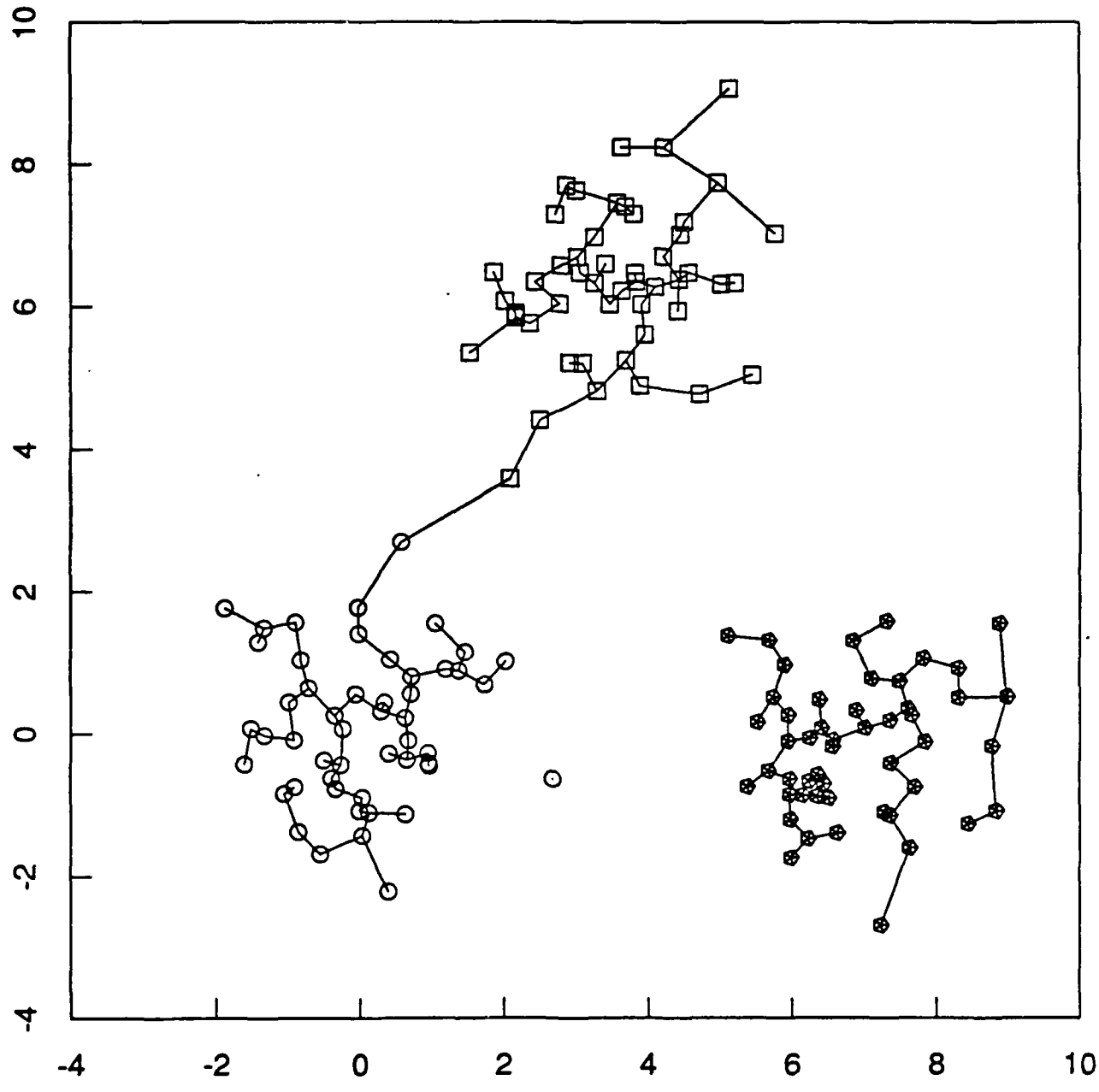


EXHIBIT 5: A poor maxrank solution (49 total minorities) for t = 7. The reason for the failure is clear: the lone point at the center of the picture produces links with very high maxranks. In the full spanning vine, this point was joined to two other points by links with maxranks of 12 and 41.

EXHIBIT 6: *Some sample results on total minorities for the basic criterion algorithm.*

TOTAL MINORITIES			
	t=5*	*t*=4.5**	*t*=4***
Median	5h	13	21h
uHinge	13	26	31
Max	27	44	40
(Mean)	9.3	17.5	21.0

* Actual total minorities: 1,4,4,5,5,6,7,13,21,27

** Actual total minorities: 3,3,10,11,12,14,24,26,28,44

*** Actual total minorities: 6,8,8,11,21,22,27,31,36,40

Note: For "h" see note to Exhibit 4.

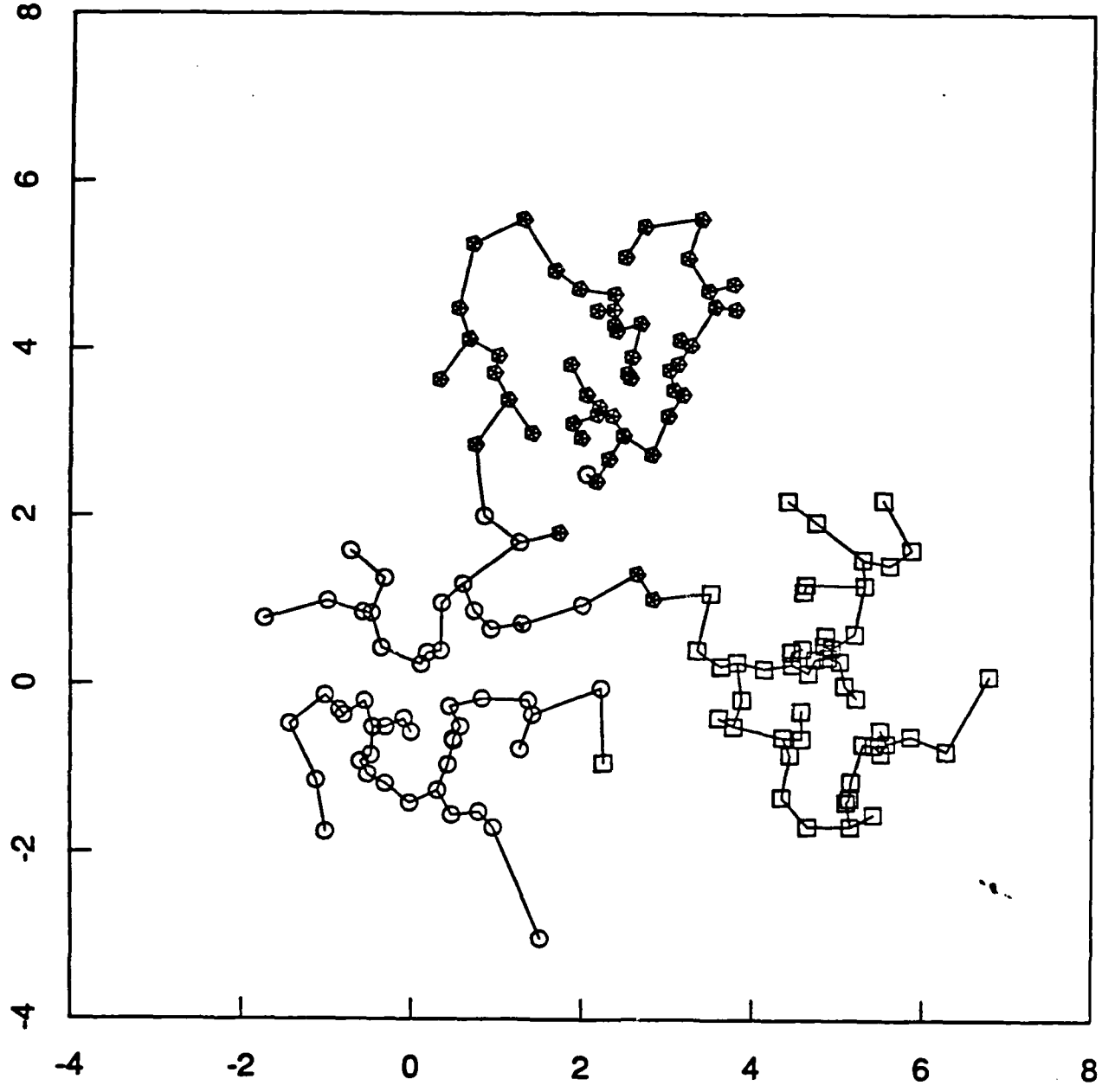


EXHIBIT 7: The worst of ten solutions for t=4.5 by the basic criterion algorithm, this configuration yields 44 total minorities. Note the filament running through the center of the picture connecting the square and circle subswarms, through two points of the third subswarm. In the central region between the two subswarms, no branches split off from the filament.

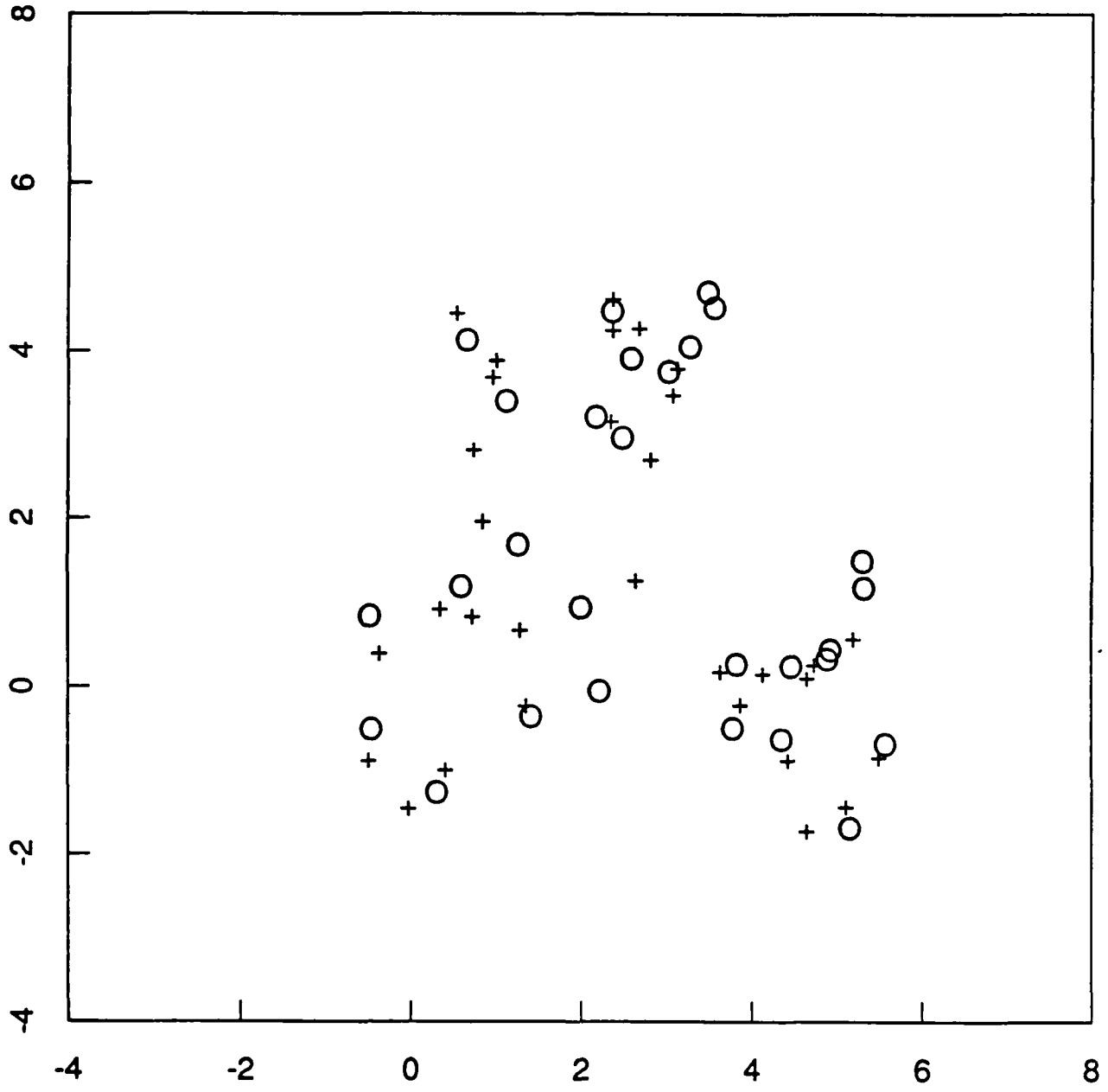


EXHIBIT 8: The high-order (circles) and vine-neighbor (crosses) core points for the data of Exhibit 7. As suspected, the core points appear to be separated into three groups. Note that the high-order points alone give a cleaner separation; it is this observation that leads us to our second definition of core points.

EXHIBIT 9: Minority counts for the basic core-cluster technique using the first type of core points.

TOTAL MINORITIES		
	t=4.5*	t=4**
Median	17	16h
uHinge	23	31
Max	53	34
(Mean)	18.0	19.3

* Actual total minorities: 3,6,6,9,16,18,21,23,25,53

** Actual total minorities: 5,9,9,9,10,23,31,31,32,34

Note: For "h" see note to Exhibit 4.

EXHIBIT 10: Total minorities for the basic core-cluster technique using the second type of core points. Note the improvement over results from the first type of core points (summarized in Exhibit 9.)

TOTAL MINORITIES		
	t=4.5*	t=4**
Median	9h	10
uHinge	13	23
Max	21	29
(Mean)	10.8	13.2

* Actual total minorities: 3,3,7,8,8,11,13,13,21,21

** Actual total minorities: 4,5,6,8,9,11,12,23,25,29

Note: For "h" see note to Exhibit 4.

EXHIBIT 11: Minority counts for the core-cluster technique with core points identified by the local density algorithm.

TOTAL MINORITIES		
	t=4.5*	t=4**
Median	8	11h
uHinge	10	20
Max	14	44
(Mean)	8.1	15.4

* Actual total minorities: 3,3,6,6,8,8,10,10,13,14

** Actual total minorities: 5,6,8,9,11,12,13,20,26,44

Note: For "h" see note to Exhibit 4.

EXHIBIT 12: Comparison of local density and high-order algorithms at t=3.2 and 3.7.

TOTAL MINORITIES				
	high-order		local density	
	t=3.7*	t=3.2**	t=3.7***	t=3.2****
Median	23h	34h	20	28
uHinge	38	47	31	38
Max	43	59	47	51
(Mean)	27.7	38.2	22.6	27.5

* Actual total minorities: 17,18,19,21,22,25,36,38,38,43

** Actual total minorities: 22,31,31,34,34,35,41,47,48,59

*** Actual total minorities: 7,11,16,17,19,21,25,31,32,47

**** Actual total minorities: 12,12,16,19,25,31,32,38,39,51

Note: For "h" see note to Exhibit 4.

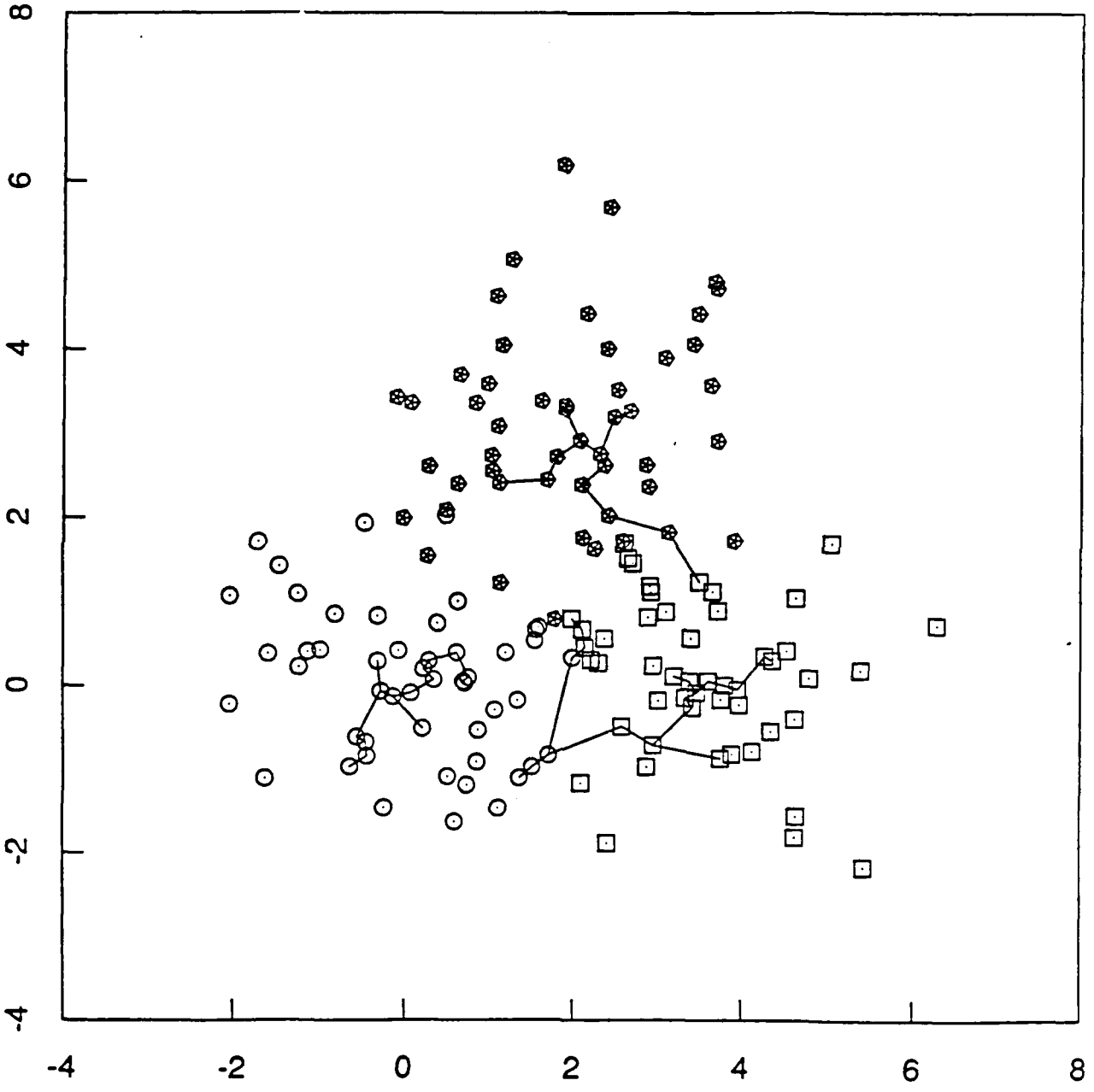


EXHIBIT 13: Core point vines for t = 3.7. The core points were identified using the local density algorithm. Note that the clusters are centered appropriately, yet branch outwards, bringing foreign points into two of the clusters. This solution gave 32 total minorities.

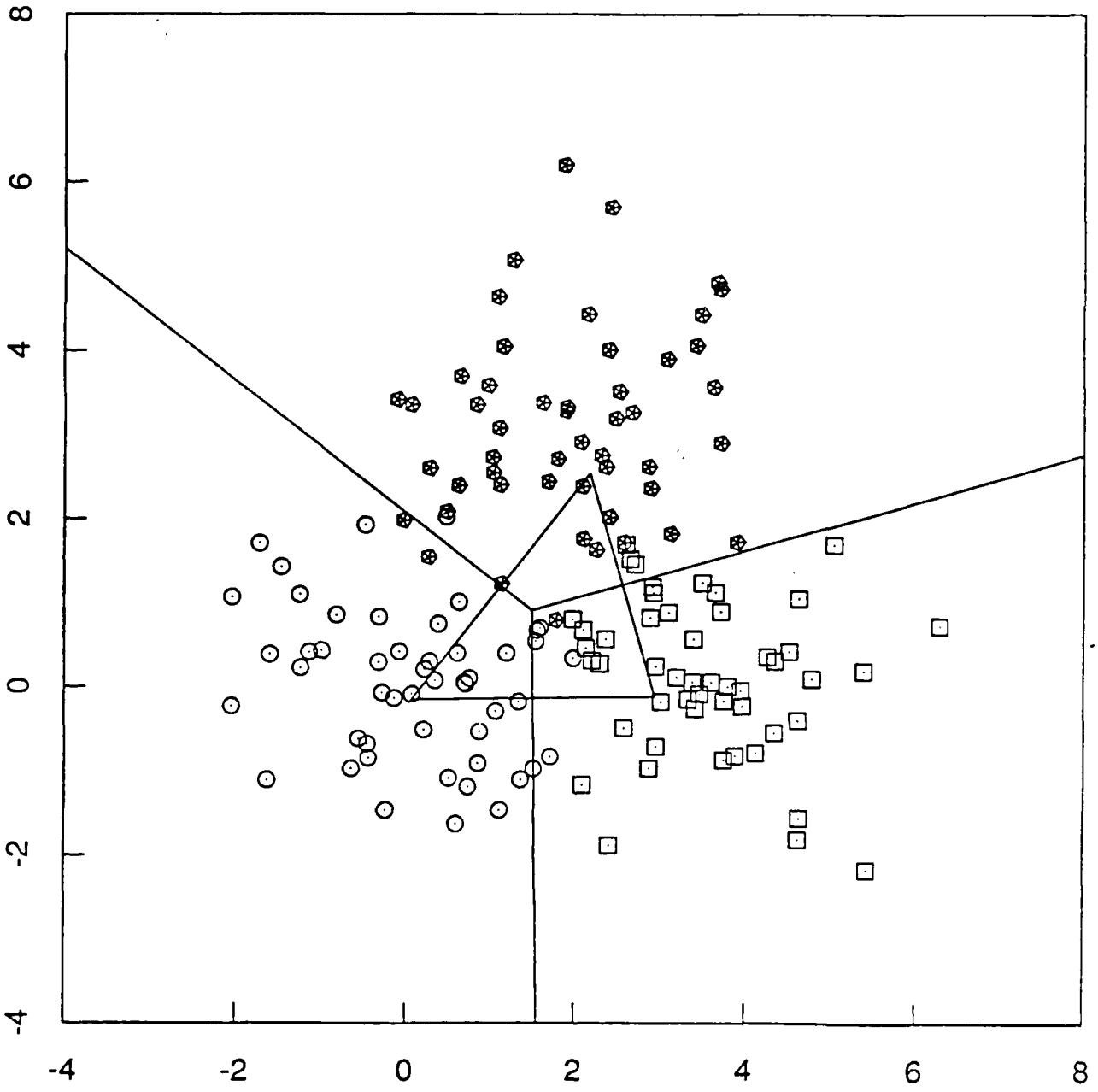


EXHIBIT 14: The triangle formed by the three centers of gravity for the data of Exhibit 13. The perpendicular bisectors, as drawn, separate the three output clusters. The number of minorities has been reduced by well over one-half, to twelve.

EXHIBIT 15: Minority counts for the basic biscetor polish and iterated bisector polish algorithms for t=2.9. Note that performance is better than that of the first core cluster techniques at t=3.2, as summarized in Exhibit 12.

TOTAL MINORITIES (t = 2.9)				
	high-order		local density	
	basic*	iterated**	basic***	iterated****
Median	19h	18	22	20
uHinge	28	21	25	22
Max	60	59	59	60
(Mean)	24.5	22.2	25.9	24.1

* Actual total minorities: 16,17,18,19,19,20,20,28,28,60

** Actual total minorities: 15,15,15,18,18,18,18,21,25,59

*** Actual total minorities: 18,19,19,20,22,22,24,25,31,59

**** Actual total minorities: 16,17,18,19,20,20,21,22,28,60

Note: For "h" see note to Exhibit 4.

EXHIBIT 16: Results obtained when the bisector polish algorithm is applied using the (non-observed) population centers of gravity at t=2.9.

TOTAL MINORITIES (t = 2.9)	
population bisectors	
Median	19
uHinge	20
Max	25
(Mean)	18.8

Actual total minorities: 14,16,17,18,19,19,19,20,21,25

EXHIBIT 17: Comparative performance for t = 2.7.

TOTAL MINORITIES (t = 2.7)			
	current*	average-linkage**	complete-linkage***
Median	25h	47h	35
uHinge	33	65	39
Max	73	97	56
(Mean)	29.7	51.1	36.0

* Actual total minorities: 16,18,20,23,25,26,29,33,34,73

** Actual total minorities: 22,23,24,25,35,60,64,65,96,97

*** Actual total minorities: 23,24,26,33,34,36,38,39,51,56

**** Already for t = 4.5, single-linkage gave a median number misclassified of 98.

Note: For "h" see note to Exhibit 4.

EXHIBIT 18: Minority counts for iterated bisector polish algorithms with 5-dimensional data.

TOTAL MINORITIES						
	t=3.2		t=2.9		t=2.7	
	high-order*	local dens**	high-order***	local dens****	high-order*****	local dens******
Median	18h	20	22	42h	26	46h
uHinge	22	29	24	46	42	56
Max	68	61	41	61	62	62
(Mean)	23.9	28.3	23.9	42.1	32.5	45.1

* Actual total minorities: 11,11,14,15,18,19,19,22,42,68

** Actual total minorities: 13,17,17,18,19,21,27,29,61,61

*** Actual total minorities: 19,19,19,21,21,23,23,24,29,41

**** Actual total minorities: 24,29,31,40,41,44,45,46,60,61

***** Actual total minorities: 19,21,21,25,26,26,38,42,45,62

****** Actual total minorities: 19,29,37,45,45,48,50,56,60,62

Note: For "h" see note to Exhibit 4.

EXHIBIT 19: Results for iterated bisector algorithms with unequal variance test beds.

TOTAL MINORITIES				
	high-order		local density	
	λ=3*	λ=2.7**	λ=3***	λ=2.7****
Median	13	17h	14	18h
uHinge	17	21	16	24
Max	22	61	53	61
(Mean)	13.9	25.1	17.9	22.8

* Actual total minorities: 7,11,11,11,12,14,16,17,18,22

** Actual total minorities: 10,14,16,16,17,18,20,21,58,61

*** Actual total minorities: 8,11,12,12,13,15,15,16,24,53

**** Actual total minorities: 12,14,15,16,17,20,22,24,27,61

Note: For "h" see note to Exhibit 4.

EXHIBIT 20: Results from using maximum likelihood and the population bisectors to classify the unequal variance data. The maximum likelihood method used the population means and variances; the bisector method used only the means.

TOTAL MINORITIES				
	max. likelihood		pop. bisectors	
	λ=3*	λ=2.7**	λ=3***	λ=2.7****
Median	12h	16	13	17
uHinge	15	19	15	18
Max	18	23	22	24
(Mean)	12.6	16.4	13.5	16.6

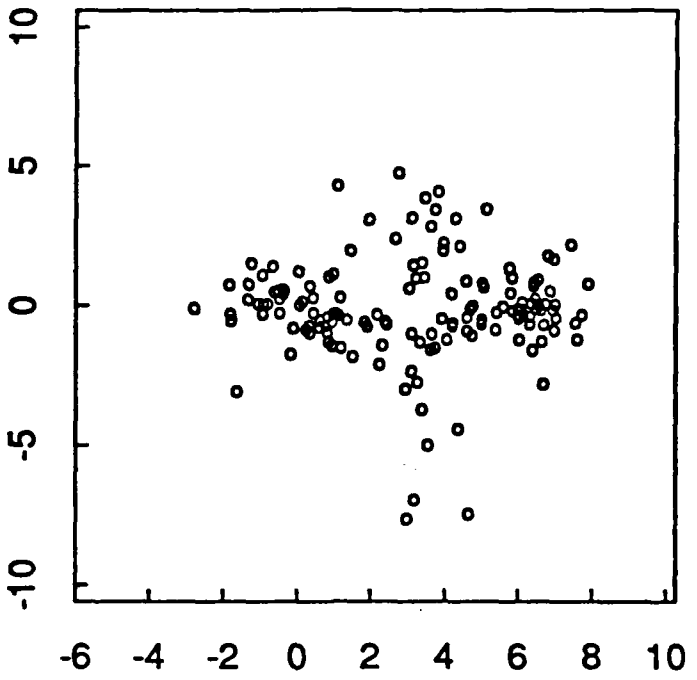
* Actual total minorities: 8,9,10,11,12,13,13,15,17,18

** Actual total minorities: 10,13,14,14,16,16,17,19,22,23

*** Actual total minorities: 7,9,11,11,12,14,14,15,20,22

**** Actual total minorities: 10,12,15,16,17,17,18,18,19,24

Note: For "h" see note to Exhibit 4.



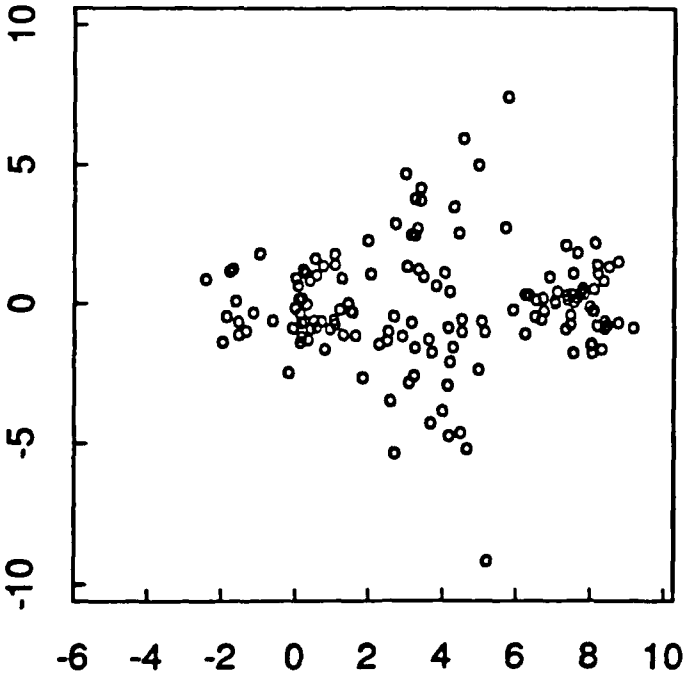


EXHIBIT 21: Two "typical" realizations of the unequal-shape test beds. The upper panel was generated with t=3.2 and gave 30 total minorities with both of the iterated bisector algorithms. The lower panel (t=3.7) gave 9 total minorities with the local density and iterated bisector algorithm and 11 total minorities with the high-order and iterated bisector algorithm.

EXHIBIT 22: Total minorities for the iterated bisector algorithms with the unequal-shape test beds.

TOTAL MINORITIES				
	high-order		local density	
	t=3.7*	t=3.2**	t=3.7***	t=3.2****
Median	13	31	10h	30
uHinge	24	34	17	33
Max	34	50	27	50
(Mean)	15.8	29.3	13.2	29.4

* Actual total minorities: 2,7,7,11,11,15,17,24,30,34

** Actual total minorities: 10,14,22,30,30,32,34,34,37,50

*** Actual total minorities: 6,8,9,9,9,12,16,17,19,27

**** Actual total minorities: 13,16,25,28,30,30,32,33,37,50

Note: For "h" see note to Exhibit 4.

APPENDIX - Code for Algorithms
(by Katherine M. Hansen)

All the clustering programs are written in the "C" language. We provide code for all the high-order and local density core-point algorithms below. The printout is divided into 6 files, formatted as follows:

1) "defs.c" (2 pages)
   This file contains all of the type definitions and variable declarations required by the programs.
2) "hiormain.c" (1 page) - basic high-order algorithm

   "densmain.c" (1 page) - basic local density algorithm

   "hoibmain.c" (1 page) - high-order and (basic & iterated) bisector algorithms

   "ldibmain.c" (1 page) - local density and (basic & iterated) bisector algorithms

   These are the main modules for each of the 4 core-point programs.
3) "modules.c" (10 pages)

   This file contains the modules (functions) called by the main modules.

To run one of the algorithms on a particular data set, the user should:

1) Copy the data into the file "datfile". Each line of the file should contain exactly one entry. The first entry should be the first variable for the first data point. The second entry is the second variable, first data point, etc. The last line of "datfile" should be the last variable of the last data point.
2) Edit the first 8 lines of "defs.c", so that NROW, NCOL, NCLUST, NCORE, and NN are set appropriately. (For the high-order algorithms, NCORE and NN may be set arbitrarily.)
3) In order, join the files "defs.c", the desired main module, and "modules.c". On a UNIX operating system, the command would be, for example:

   ```
   cat defs.c hiormain.c modules.c >  hiorder.c
   ```

   The required source code is now held in the file "hiorder.c".
4) Compile the source code, to generate an executable file. On a UNIX system, the appropriate command would be:

   ```
   cc hiorder.c -lm
   ```

   The executable code would be contained in a file a.out.
5) Run the executable program. The classifications will be placed in a file called "countfile". Additional information may be placed in other files, as follows:

"edgefile" (all 4 programs). Information on the NROW-1 edges of the complete spanning vine. The 2 endpoints, minrank, maxrank, criterion, and minsize of each edge are placed in the file.

"corefile" (all 4 programs). Information on the NCORE-NCLUST edges of the core point vines. For each edge the 6 values just described above are placed in the file.

"noncorefile" (hior and dens). Information on the NROW-numcore edges linking non-core points to core point vines. For each edge, the 6 values described above are placed in the file.

"gravfile" (hoib and ldib). Coordinates of the (iterated) centers of gravity of the core point clusters. Each row of the file contains the coordinates of the core center.

```
/* defs.c
   type and variable definitions for "C" programs */

# include <stdio.h>
# include <math.h>
#define NROW 150      /* number of data points */
#define NCOL 2        /* dimension of data */
#define NCLUST 3      /* number of clusters desired */
#define NCORE 50      /* number core points desired, for density algorithms */
#define NN 10         /* neighbor used to determine density */
#define MAXINCLUDE 15
     /* MAXINCLUDE is number of nearest neighbors of each point to keep as
        candidate edges; using MAXINCLUDE = 15 will save run-time, but
        for large data sets with well-defined clusters, MAXINCLUDE may
        need to be increased */
#define LENGTH (NROW*MAXINCLUDE)/2

typedef double DATA[NCOL];

typedef struct {
     double distto;
     int ptnum;
}  DIST, DISTANCES[NROW];

typedef struct {
     int num;
     DISTANCES dat;
} DISTMAT;

typedef struct {
     int  ptnum;
     DATA coord;
}  POINT;

typedef struct {
     int start;
     int end;
     int minrank;
     int birankplus; /* birank + 5; used in all calcs needing birank */
     double sq_dist;
     int minsize;
     double criterion;
}  EDGESTRUCT;            /* structure of candidate edges */

typedef struct {
     int lowc;
     DATA coord;
} GRAVCENT;

typedef struct {  /* cluster status of each point */
      int lowc;     /* lowest numberes point contained in same cluster */
      int sizec;    /* number of points in cluster */
      int order;    /* order of point, after preliminary run */
      int corept;   /* indicates whether or not a core point */
} CLUSTINFO;

typedef struct {
     int ptnum;
     double dist;
}  DENSINFO;

FILE *datfp,                                          /* input file */
     *countfp, *corefp, *edgefp, *gravfp, *noncorefp,    /* output files */
     *fopen();
int locmaxct, numcore, truesize,
    connection[NROW][NROW], locmax[NROW],
```

```
    rankdat[NROW][NROW], newrankdat[NROW][NROW],
    logdens_sort(), sort_rows();
double logvect[NROW],  logdens[NROW], log();
CLUSTINFO lowest[NROW];
DENSINFO densdist[NROW];
DISTMAT distdat[NROW], newdistdat[NROW];
EDGESTRUCT edgedat[LENGTH], newedgedat[LENGTH];
GRAVCENT center[NCLUST], newcent[NCLUST];
POINT pointdat[NROW], newptdat[NROW];
```

```
/* hiormain.c
   main module for hiorder algorithm */

main()

{

datfp = fopen("datfile", "r");
countfp = fopen("countfile","w");
corefp = fopen("corefile","w");
edgefp = fopen("edgefile","w");
noncorefp = fopen("noncorefile","w");
initialize_data_structures();
reset_lowest(1);
find_dists(pointdat, distdat, NROW);
first_sort(distdat, rankdat, NROW);
load_edgemat(distdat, rankdat, edgedat, &truesize, NROW, pointdat);
grow_vine(0, edgedat, truesize, NROW-1);
find_hiorder_points(&numcore);
reset_lowest(0);
find_dists(newptdat, newdistdat, numcore);
first_sort(newdistdat, newrankdat, numcore);
load_edgemat(newdistdat, newrankdat, newedgedat, &truesize, numcore, newptdat);
grow_vine(1, newedgedat, truesize, numcore-NCLUST);
load_noncore_edgemat(&truesize);
grow_vine(2, edgedat, truesize, NROW-numcore);
print_classification();
}
```

```
/* densmain.c
   main module for the local density algorithm */

main()

{

datfp = fopen("datfile","r");
countfp = fopen("countfile","w");
corefp = fopen("corefile","w");
edgefp = fopen("edgefile","w");
noncorefp = fopen("noncorefile","w");
initialize_data_structures();
reset_lowest(1);
find_dists(pointdat, distdat, NROW);
first_sort(distdat, rankdat, NROW);
load_edgemat(distdat, rankdat, edgedat, &truesize, NROW, pointdat);
reset_matrix(connection);
grow_vine(0, edgedat, truesize, NROW-1);
smooth_densities();
find_local_maxes(&locmaxct);
find_hi_dens_points();
qsort((char *) densdist, NROW, sizeof(DENSINFO), logdens_sort);
make_newptdat(&numcore);
reset_lowest(0);
find_dists(newptdat, newdistdat, numcore);
first_sort(newdistdat, newrankdat, numcore);
load_edgemat(newdistdat, newrankdat, newedgedat, &truesize, numcore, newptdat);
reset_matrix(connection);
grow_vine(1, newedgedat, truesize, numcore-NCLUST);
load_noncore_edgemat(&truesize);
grow_vine(2, edgedat, truesize, NROW-numcore);
print_classification();
}
```

```
/* hoibmain.c
   main module for the hiorder and bisector and
   hiorder and iterated bisector algorithms */

main()

{

datfp = fopen("datfile", "r");
countfp = fopen("countfile","w");
corefp = fopen("corefile","w");
edgefp = fopen("edgefile","w");
gravfp = fopen("gravfile","w");
initialize_data_structures();
reset_lowest(1);
find_dists(pointdat, distdat, NROW);
first_sort(distdat, rankdat, NROW);
load_edgemat(distdat, rankdat, edgedat, &truesize, NROW, pointdat);
grow_vine(0, edgedat, truesize, NROW-1);
find_hiorder_points(&numcore);
reset_lowest(0);
find_dists(newptdat, newdistdat, numcore);
first_sort(newdistdat, newrankdat, numcore);
load_edgemat(newdistdat, newrankdat, newedgedat, &truesize, numcore, newptdat);
grow_vine(1, newedgedat, truesize, numcore-NCLUST);
reset_centers(center);
find_gravity_centers();
gravity_center_classify(center);
print_classification();
reset_centers(newcent);
find_iterated_grav_centers();
gravity_center_classify(newcent);
print_classification();
}
```

```
/* ldibmain.c
   main module for local density and bisector and
   local density and iterated bisector algorithms */

main()

{
datfp = fopen("datfile", "r");
countfp = fopen("countfile", "w");
corefp = fopen("corefile", "w");
edgefp = fopen("edgefile", "w");
gravfp = fopen("gravfile", "w");
initialize_data_structures();
reset_lowest(1);
find_dists(pointdat, distdat, NROW);
first_sort(distdat, rankdat, NROW);
load_edgemat(distdat, rankdat, edgedat, &truesize, NROW, pointdat);
reset_matrix(connection);
grow_vine(0, edgedat, truesize, NROW-1);
smooth_densities();
find_local_maxes(&locmaxct);
find_hi_dens_points();
qsort((char *) densdist, NROW, sizeof(DENSINFO), logdens_sort);
make_newptdat(&numcore);
reset_lowest(0);
find_dists(newptdat, newdistdat, numcore);
first_sort(newdistdat, newrankdat, numcore);
load_edgemat(newdistdat, newrankdat, newedgedat, &truesize, numcore, newptdat);
reset_matrix(connection);
grow_vine(1, newedgedat, truesize, numcore-NCLUST);
reset_centers(center);
find_gravity_centers();
gravity_center_classify(center);
print_classification();
reset_centers(newcent);    /* iterating bisectors */
find_iterated_grav_centers();
gravity_center_classify(newcent);
print_classification();
}
```

```
/* modules.c
   modules of "C" code used in each of the clustering programs */

initialize_data_structures()
/* initializes pointdat and logvect structures */

{
int row,col;

for (row = 0; row < NROW; row++)  {
     pointdat[row].ptnum = row;
     for (col = 0; col < NCOL; col++)
          (void) fscanf(datfp,"%f",&pointdat[row].coord[col]);
     }
logvect[0] = 1;
for (row = 1; row < NROW; row++)
     logvect[row] = log((double)row) + 1;
}


reset_lowest(reset_core)
/* resets the lowest vector */

int reset_core;
{
int row;

if (reset_core == 1)        /* first run; set all points as non-core */
     for (row = 0; row < NROW; row++) {
          lowest[row].lowc = row;
          lowest[row].sizec = 1;
          lowest[row].order = 0;
          lowest[row].corept = 0;
          }
                            /* second run; leave core points indicated as such */
else for (row = 0; row < NROW; row++)  {
          lowest[row].lowc = row;
          lowest[row].sizec = 1;
          lowest[row].order = 0;
          }
}


find_dists(pointmat, distmat, size)
/* finds euclidean distance from each point to each other */

POINT pointmat[NROW];
DISTMAT distmat[NROW];
int size;
{
int row,col,k;
double tempdist;

for (row = 0; row < size; row++) {
     for (col = row ; col < size; col++)
          if (row == col) {
               distmat[row].dat[col].distto = 0;
               distmat[row].dat[col].ptnum = row;
               }
          else {
               tempdist = 0;
               for (k = 0; k < NCOL; k++)
                    tempdist = tempdist + (pointmat[row].coord[k] -
                              pointmat[col].coord[k])*(pointmat[row].coord[k] -
                              pointmat[col].coord[k]);
               distmat[row].dat[col].distto = tempdist;
```

```
                distmat[row].dat[col].ptnum = col;
                distmat[col].dat[row].distto = distmat[row].dat[col].distto;
                distmat[col].dat[row].ptnum = row;
                }
        distmat[row].num = row;
        }
}


first_sort(distmat, rankmat, size)
/* sorts each row of distance matrix */

DISTMAT distmat[NROW];
int rankmat[NROW][NROW];
int size;
{
int col, row, temp;

for (row = 0; row < size; row++) {
        qsort((char *)distmat[row].dat,size,sizeof(DIST),sort_rows);
        for (col = 1; col < size; col++) {
                temp = distmat[row].dat[col].ptnum;
                rankmat[row][temp] = col;
                }
        }
}


load_edgemat(distmat, rankmat, edgemat, truesize, size, pointmat)
/* creates the matrix of candidate edges; includes all edges
        with minrank < MAXINCLUDE */

DISTMAT distmat[NROW];
int *truesize, rankmat[NROW][NROW], size;
EDGESTRUCT edgemat[LENGTH];
POINT pointmat[NROW];
{
int row, col, i;

i = 0;
for (row = 0; row < size; row++)
        for (col = row + 1; col < size; col++ )
                if (rankmat[row][col]<MAXINCLUDE||rankmat[col][row]<MAXINCLUDE) {
                edgemat[i].start = pointmat[row].ptnum;
                edgemat[i].end = pointmat[col].ptnum;
                if (rankmat[row][col] < rankmat[col][row]) {
                        edgemat[i].minrank = rankmat[row][col];
                        edgemat[i].birankplus = rankmat[col][row] + 5;
                        }
                else {
                        edgemat[i].minrank = rankmat[col][row];
                        edgemat[i].birankplus = rankmat[row][col] + 5;
                        }
                edgemat[i].sq_dist = distmat[row].dat[rankmat[row][col]].distto;
                edgemat[i].minsize = 1;
                edgemat[i].criterion = (edgemat[i].birankplus);
                i = i + 1;
                }
*truesize = i;
}


reset_matrix(mat)
/* may be used with either checked or connection matricies */

int mat[NROW][NROW];
```

```
{
int i, j;

for (i = 0; i < NROW; i++)
     for (j = 0; j < i; j++) {
          mat[i][j] = 0;
          mat[j][i] = 0;
          }
}


grow_vine(type, edgemat, truesize, numlinks)
/* finds the edges which are to be added to vine; the variable "type"
   indicates whether preliminary vine (type = 0), core point vine
   (type = 1), or final vine (type = 2) */

EDGESTRUCT edgemat[LENGTH];
int truesize, type, numlinks;
{
int numedges, numleft, newc, newsize;

numleft = truesize;
numedges = 0;
while (numedges < numlinks) {
     min_first(edgemat, numleft);
     add_edge(type, edgemat[0], &newc, &newsize);
     update_criterion(edgemat, &numleft, newc, newsize);
     numedges = numedges + 1;
     }
}


min_first(edgemat, numleft)
/* places edge with lowest criterion value in first position of edgemat */

EDGESTRUCT edgemat[LENGTH];
int numleft;
{
int row, temprow;
EDGESTRUCT tempedge;

temprow = 0;
for (row = 1; row < numleft; row++)
     if (edgemat[row].criterion < edgemat[temprow].criterion)
          temprow = row;
     else if ((edgemat[row].criterion == edgemat[temprow].criterion)&&
                    (edgemat[row].sq_dist < edgemat[temprow].sq_dist))
          temprow = row;
tempedge = edgemat[0];
edgemat[0] = edgemat[temprow];
edgemat[temprow] = tempedge;
}


update_criterion(edgemat, numleft, newc, newsize)
/* updates the "criterion" of each candidate edge after a new addition
   to the vine; deletes all circuit-forming edges. */

EDGESTRUCT edgemat[NROW];
int *numleft, newc, newsize;
{

int pos, tempint;

for (pos = 0; pos < *numleft; pos++) {
     if ((lowest[edgemat[pos].start].lowc == lowest[edgemat[pos].end].lowc)) {
```

```
            for (tempint = pos; tempint < *numleft; tempint++)
                edgemat[tempint]=edgemat[tempint + 1];
            *numleft = *numleft - 1;
            pos = pos - 1;
            }
        else if (lowest[edgemat[pos].start].lowc == newc ) {
                if (lowest[edgemat[pos].end].sizec > newsize)
                    edgemat[pos].minsize = newsize;
                else edgemat[pos].minsize = lowest[edgemat[pos].end].sizec;
                edgemat[pos].criterion =
                     (edgemat[pos].birankplus)*(logvect[edgemat[pos].minsize]);
                }
        else if (lowest[edgemat[pos].end].lowc == newc ) {
            if (lowest[edgemat[pos].start].sizec > newsize)
                edgemat[pos].minsize = newsize;
            else edgemat[pos].minsize = lowest[edgemat[pos].start].sizec;
            edgemat[pos].criterion =
                    (edgemat[pos].birankplus) *(logvect[edgemat[pos].minsize]);
            }
        }
}


add_edge(type, edge, newc, newsize)
/* adds edges to vine, writes them in order in appropriate file */

EDGESTRUCT edge;
int type, *newc, *newsize;
{
int col, tempint;

if (type == 0)
fprintf(edgefp,"%5d %5d %5d %5d %f %5d\n",edge.start, edge.end, edge.minrank,
        edge.birankplus - 5, edge.criterion, edge.minsize);
else if (type == 1) {
fprintf(corefp,"%5d %5d %5d %5d %f %5d\n",edge.start, edge.end, edge.minrank,
        edge.birankplus - 5, edge.criterion, edge.minsize);
        fflush(corefp);
        }
else
fprintf(noncorefp,"%5d %5d %5d %5d %f %5d\n",edge.start, edge.end,
        edge.minrank, edge.birankplus - 5, edge.criterion, edge.minsize);
lowest[edge.start].order += 1;
lowest[edge.end].order += 1;
connection[edge.start][edge.end] = 1;
connection[edge.end][edge.start] = 1;
if (lowest[edge.start].lowc < lowest[edge.end].lowc) {
     *newc = lowest[edge.start].lowc;
     *newsize = (lowest[edge.start].sizec + lowest[edge.end].sizec);
     tempint = lowest[edge.end].lowc;
     for (col = 0; col < NROW; col++)
          if (lowest[col].lowc == tempint) {
               lowest[col].lowc = *newc;
               lowest[col].sizec = *newsize;
               }
          else if (lowest[col].lowc == *newc)
               lowest[col].sizec = *newsize;
     }
else if (lowest[edge.start].lowc > lowest[edge.end].lowc) {
     *newc = lowest[edge.end].lowc;
     *newsize = (lowest[edge.start].sizec + lowest[edge.end].sizec);
     tempint = lowest[edge.start].lowc;
     for (col = 0; col < NROW; col++)
          if (lowest[col].lowc == tempint) {
               lowest[col].lowc = *newc;
               lowest[col].sizec = *newsize;
```

```
                }
            else if (lowest[col].lowc == *newc)
                lowest[col].sizec = *newsize;
        }
    }


smooth_densities()
/* smoothes densities on the vine structure */

{
int ptnum, col, tempct, changed;
double newlogdens[NROW], tempdens[10], ors2;

for (ptnum = 0; ptnum < NROW; ptnum++)
    logdens[ptnum] = -2*(log(distdat[ptnum].dat[NN].distto));
changed = 1;
while (changed == 1) {
    changed = 0;
    for (ptnum = 0; ptnum < NROW; ptnum++)
        if (lowest[ptnum].order == 1)
            newlogdens[ptnum] = logdens[ptnum];
        else {
            tempdens[0] = logdens[ptnum];
            tempct = 1;
            for (col = 0; col < NROW; col++)
                if (tempct <= lowest[ptnum].order)
                    if (connection[ptnum][col] == 1) {
                        tempdens[tempct] = logdens[col];
                        tempct += 1;
                        }
            find_orstat2(tempdens, &ors2, tempct);
            newlogdens[ptnum] = ors2;
            }
    for (ptnum = 0; ptnum < NROW; ptnum++) {
        if (logdens[ptnum] != newlogdens[ptnum])
            changed = 1;
        logdens[ptnum] = newlogdens[ptnum];
        }
    }
}


find_orstat2(tempdens, ors2, order)
/* finds second-highest element of tempdens */

double tempdens[10], *ors2;
int order;
{
int row, maxpos, pos2;

maxpos = 0;
for (row = 1; row < order; row++)
    if (tempdens[row] > tempdens[maxpos])
        maxpos = row;
if (maxpos == 0)
    pos2 = 1;
else pos2 = 0;
for (row = 0; row < order; row++)
    if ((tempdens[row] > tempdens[pos2]) && (row != maxpos))
        pos2 = row;
*ors2 = tempdens[pos2];
}


find_local_maxes(locmaxct)
```

```
/* identifies the local maxima on the vine */

int *locmaxct;
{
int row, col, maxpos;

for (row = 0; row < NROW; row++) {
    maxpos = row;
    for (col = 0; col < NROW; col++)
         if (connection[row][col] == 1)
              if (logdens[col] > logdens[row])
                   maxpos = col;
    if (maxpos == row) {
         locmax[*locmaxct] = row;
         *locmaxct += 1;
         }
    }
}


find_hi_dens_points()
/* computes local density of each point */

{
int row, col, max, checked[NROW][NROW];

for (row = 0; row < NROW; row++)  {
     densdist[row].ptnum = row;
     densdist[row].dist = 0;
     }
for (row = 0; row < locmaxct; row++) {
     reset_matrix(checked);
     max = locmax[row];
     for (col = 0; col < NROW; col++)
          if (connection[max][col] == 1) {
               checked[max][col] = 1;
               checked[col][max] = 1;
               follow_branch(max, max, col, checked);
               }
     }
}


follow_branch(curmax, pt1, pt2, checked)
/* recursive procedure for moving about vine structure */

int curmax, pt1, pt2, checked[NROW][NROW];
{
int num, conpts[NROW], tempct;
double diff;

tempct = 0;
if (logdens[pt2] <= logdens[pt1]) {
     diff = logdens[curmax] - logdens[pt2];
     if (diff > densdist[pt2].dist)
          densdist[pt2].dist = diff;
     for (num = 0; num < NROW; num++)
          if ((connection[pt2][num] == 1) && (checked[pt2][num] == 0)) {
               checked[pt2][num] = 1;
               checked[num][pt2] = 1;
               conpts[tempct] = num;
               tempct += 1;
               }
     for (num = 0; num < tempct; num++)
          follow_branch(curmax, pt2, conpts[num], checked);
     }
```

```
}


logdens_sort(dens1,dens2)
/* sort function for qsort */

DENSINFO *dens1, *dens2;
{

if ((*dens1).dist < (*dens2).dist)
     return(-1);
else if ((*dens1).dist > (*dens2).dist)
     return(1);
else return(0);
}


make_newptdat(numcore)
/* makes new data matrix, containing only core points */

int *numcore;
{
int row;

for (row = 0; row < NCORE; row++)  {
     newptdat[row] = pointdat[densdist[row].ptnum];
     lowest[densdist[row].ptnum].corept = 1;
     }
row = NCORE-1;
*numcore = NCORE;
while (densdist[row].dist == densdist[NCORE].dist) {
     lowest[densdist[row].ptnum].corept = 0;
     row -= 1;
     *numcore -= 1;
     }
fprintf(countfp, "numcore = %d\n", *numcore);
}


find_gravity_centers()
/* finds gravity center of each core-point cluster */

{
int row, col, dim, num;

num = 0;
for (row = 0; row < NROW; row++)
     if ((lowest[row].lowc == row) && (lowest[row].corept == 1)) {
          for (col = 0; col < NROW; col++)
               if (lowest[col].lowc == row)
                    for (dim = 0; dim < NCOL; dim++)
                         center[num].coord[dim] = center[num].coord[dim] +
                                                  pointdat[col].coord[dim];
          for (dim = 0; dim < NCOL; dim++)
               center[num].coord[dim]=center[num].coord[dim]/lowest[row].sizec;
          center[num].lowc = row;
          num += 1;
          }
fprintf(countfp,"Initial classification:\n");
}


gravity_center_classify(cent)
/* classifies points based on nearest center of gravity */

GRAVCENT cent[NCLUST];
```

```
{
int row, mink, col, k;
double tempmin, tempdist;

for (row = 0; row < NROW; row++)  {
     mink = 0;
     tempmin = 0.0;
     for (col = 0; col < NCOL; col++)
          tempmin = tempmin + (pointdat[row].coord[col] -
                    cent[0].coord[col])*(pointdat[row].coord[col] -
                    cent[0].coord[col]);
     for (k = 1; k < NCLUST; k++)  {
          tempdist = 0.0;
          for (col = 0; col < NCOL; col++)
          tempdist = tempdist + (pointdat[row].coord[col] -
                    cent[k].coord[col])*(pointdat[row].coord[col] -
                    cent[k].coord[col]);
          if (tempdist < tempmin)  {
               tempmin = tempdist;
               mink = k;
               }
          }
     lowest[row].lowc = cent[mink].lowc;
     }
}


find_iterated_grav_centers()
/* finds iterated centers of gravity */

{
int row, col, dim, count;

for (row = 0; row < NCLUST; row++) {
     newcent[row].lowc = center[row].lowc;
     count = 0;
     for (col = 0; col < NROW; col++)
          if (lowest[col].lowc == center[row].lowc) {
               count += 1;
               for (dim = 0; dim < NCOL; dim++)
                    newcent[row].coord[dim] = newcent[row].coord[dim] +
                                              pointdat[col].coord[dim];
               }
     for (col = 0; col < NCOL; col++)
          newcent[row].coord[col] = newcent[row].coord[col]/count;
     for (col = 0; col < NCOL; col++)
          fprintf(gravfp,"%f ",newcent[row].coord[col]);
     fprintf(gravfp, "\n");
     }
fprintf(countfp,"\n");
fprintf(countfp,"Classification after iteration:\n");
}


reset_centers(cent)
/* re-initializes center structure; used on both center and newcent */

GRAVCENT cent[NCLUST];
{
int row,col;

for (row = 0; row < NCLUST; row++)
     for (col = 0; col < NCOL; col++)
          cent[row].coord[col] = 0;
}
```

```
find_hiorder_points(numcore)
/* designates points of order > 2 as core points */

int *numcore;
{
int row, tempnum;

tempnum = 0;
for (row = 0; row < NROW; row++)
     if (lowest[row].order > 2) {
          newptdat[tempnum] = pointdat[row];
          lowest[row].corept = 1;
          tempnum += 1;
          }
*numcore = tempnum;
}


sort_rows(dist1,dist2)
/* sort function for qsort */

DIST *dist1, *dist2;
{

if ((*dist1).distto > (*dist2).distto)
     return(1);
else if ((*dist1).distto < (*dist2).distto)
     return(-1);
else return (0);
}


load_noncore_edgemat(truesize)
/* creates the matrix of candidate edges for non-core points */

int *truesize;
{
int row,col,i;

i = 0;
for (row = 0; row < NROW; row++)
     for (col = row + 1; col < NROW; col++ )
          if (((lowest[col].corept != 1)||(lowest[row].corept != 1))&&
               ((rankdat[row][col] < MAXINCLUDE)||
                (rankdat[col][row] < MAXINCLUDE)))    {
               edgedat[i].start = row;
               edgedat[i].end = col;
               if (rankdat[row][col] < rankdat[col][row]) {
                    edgedat[i].minrank = rankdat[row][col];
                    edgedat[i].birankplus = rankdat[col][row] + 5;
                    }
               else {
                    edgedat[i].minrank = rankdat[col][row];
                    edgedat[i].birankplus = rankdat[row][col] + 5;
                    }
               edgedat[i].sq_dist =
                              distdat[row].dat[rankdat[row][col]].distto;
               edgedat[i].minsize = 1;
               if (((lowest[col].sizec == 1)&&(lowest[row].sizec == 1))&&
                   ((lowest[col].corept != 1)&&(lowest[row].corept != 1)))
                    edgedat[i].criterion = NROW + 6;
     /* above prevents links between two single noncore points */
               else edgedat[i].criterion = edgedat[i].birankplus;
               i = i + 1;
               }
```

```
*truesize = i;
}


print_classification()
/* prints classification results in file "countfile" */

{
int row;

for (row = 0; row < NROW; row++)
     fprintf(countfp, "point # %d in cluster %d\n", row, lowest[row].lowc);
fprintf(countfp, "\n");
}
```

END

DATE

FILMED

5-88

DTIC